

# cs280s21-b.sg

[Dashboard](#) / My courses / [cs280s21-b.sg](#) / [General](#) / [Assignment 1: Object Allocator](#)

[Description](#)

[Submission](#)

[Edit](#)

Submission view

Submitted on Friday, January 29, 2021, 7:39 PM ([Download](#))

Automatic evaluation [-]

**Proposed grade: 100 / 100**

**Compilation** [-]

Running your code.....

Executable successfully created

**Comments** [-]

[±] **Summary of tests**

ObjectAllocator.h

```
1 //-----
2 #ifndef OBJECTALLOCATORH
3 #define OBJECTALLOCATORH
4 //-----
5
6 #include <string>
7
8 // If the client doesn't specify these:
9 static const int DEFAULT_OBJECTS_PER_PAGE = 4;
10 static const int DEFAULT_MAX_PAGES = 3;
11
12 /*!
13  * Exception class
14 */
15 class OAException
16 {
17 public:
18 /*!
19  * Possible exception codes
20 */
21 enum OA_EXCEPTION
22 {
23     E_NO_MEMORY,      //!< out of physical memory (operator new fails)
24     E_NO_PAGES,       //!< out of logical memory (max pages has been reached)
25     E_BAD_BOUNDARY,   //!< block address is on a page, but not on any block-boundary
26     E_MULTIPLE_FREE,  //!< block has already been freed
27     E_CORRUPTED_BLOCK //!< block has been corrupted (pad bytes have been overwritten)
28 };
29
30 /*!
31  * Constructor
32 */
33 \param ErrorCode
34     One of the 5 error codes listed above
35
36 \param Message
37     A message returned by the what method.
38 */
39 OAException(OA_EXCEPTION ErrorCode, const std::string& Message) : error_code_(ErrorCode), message_(Message) {};
40
41 /*!
42  * Destructor
43 */
44 virtual ~OAException() {}
45
46 /*!
47  * Retrieves the error code
48 */
49
50 \return
51     One of the 5 error codes.
52 */
53 OA_EXCEPTION code() const {
54     return error_code_;
55 }
56
57 /*!
58  * Retrieves a human-readable string regarding the error.
59 */
60
61 \return
62     The NUL-terminated string representing the error.
63 */
64 virtual const char *what() const {
65     return message_.c_str();
66 }
67 private:
68 OA_EXCEPTION error_code_; //!< The error code (one of the 5)
69 std::string message_;    //!< The formatted string for the user.
70 };
71
72 /*!
73  * ObjectAllocator configuration parameters
74 */
75 struct OAConfig
76 {
77     static const size_t BASIC_HEADER_SIZE = sizeof(unsigned) + 1; //!< allocation number + flags
78     static const size_t EXTERNAL_HEADER_SIZE = sizeof(void*);      //!< just a pointer
79
80 /*!
81  * The different types of header blocks
82 */
83 enum HBLOCK_TYPE{hbNone, hbBasic, hbExtended, hbExternal};
84
85 /*!
86  * POD that stores the information related to the header blocks.
87 */
88 struct HeaderBlockInfo
89 {
90     HBLOCK_TYPE type_; //!< Which of the 4 header types to use?
91     size_t size_;       //!< The size of this header
92     size_t additional_; //!< How many user-defined additional bytes
93
94 /*!
95  * Constructor
96 */
97 \param type
98     The kind of header blocks in use.
99
100 \param additional
101     The number of user-defined additional bytes required.
102 */
103 HeaderBlockInfo(HBLOCK_TYPE type = hbNone, unsigned additional = 0) : type_(type), size_(0), additional_(additional)
104 {
105     if (type_ == hbBasic)
106         size_ = BASIC_HEADER_SIZE;
```

```

108
109     else if (type_ == hbExtended) // alloc # + use counter + flag byte + user-defined
110         size_ = sizeof(unsigned int) + sizeof(unsigned short) + sizeof(char) + additional_;
111     else if (type_ == hbExternal)
112         size_ = EXTERNAL_HEADER_SIZE;
113     };
114 };
115 */
116
117 /**
118  * \param UseCPPMemManager
119  *     Determines whether or not to by-pass the OA.
120
121  * \param ObjectsPerPage
122  *     Number of objects for each page of memory.
123
124  * \param MaxPages
125  *     Maximum number of pages before throwing an exception. A value
126  *     of 0 means unlimited.
127
128  * \param DebugOn
129  *     Is debugging code on or off?
130
131  * \param PadBytes
132  *     The number of bytes to the left and right of a block to pad with.
133
134  * \param HBInfo
135  *     Information about the header blocks used
136
137  * \param Alignment
138  *     The number of bytes to align on.
139 */
140 OAConfig(bool UseCPPMemManager = false,
141           unsigned ObjectsPerPage = DEFAULT_OBJECTS_PER_PAGE,
142           unsigned MaxPages = DEFAULT_MAX_PAGES,
143           bool DebugOn = false,
144           unsigned PadBytes = 0,
145           const HeaderBlockInfo &HBInfo = HeaderBlockInfo(),
146           unsigned Alignment = 0) : UseCPPMemManager_(UseCPPMemManager),
147           ObjectsPerPage_(ObjectsPerPage),
148           MaxPages_(MaxPages),
149           DebugOn_(DebugOn),
150           PadBytes_(PadBytes),
151           HBBlockInfo_(HBInfo),
152           Alignment_(Alignment)
153 {
154     HBlockInfo_ = HBInfo;
155     LeftAlignSize_ = 0;
156     InterAlignSize_ = 0;
157 }
158
159     bool UseCPPMemManager_;      //!<< by-pass the functionality of the OA and use new/delete
160     unsigned ObjectsPerPage_;   //!<< number of objects on each page
161     unsigned MaxPages_;        //!<< maximum number of pages the OA can allocate (0=unlimited)
162     bool DebugOn_;            //!<< enable/disable debugging code (signatures, checks, etc.)
163     unsigned PadBytes_;       //!<< size of the left/right padding for each block
164     HeaderBlockInfo HBlockInfo_; //!<< size of the header for each block (0=no headers)
165     unsigned Alignment_;      //!<< address alignment of each block
166     unsigned LeftAlignSize_;   //!<< number of alignment bytes required to align first block
167     unsigned InterAlignSize_;  //!<< number of alignment bytes required between remaining blocks
168 };
169
170 /**
171  * POD that holds the ObjectAllocator statistical info
172 */
173
174 struct OAStats
175 {
176 /**
177  * \param Constructor
178  */
179 OAStats() : ObjectSize_(0), PageSize_(0), FreeObjects_(0), ObjectsInUse_(0), PagesInUse_(0),
180           MostObjects_(0), Allocations_(0), Deallocations_(0) {};
181
182     size_t ObjectSize_;      //!<< size of each object
183     size_t PageSize_;        //!<< size of a page including all headers, padding, etc.
184     unsigned FreeObjects_;   //!<< number of objects on the free list
185     unsigned ObjectsInUse_;  //!<< number of objects in use by client
186     unsigned PagesInUse_;   //!<< number of pages allocated
187     unsigned MostObjects_;   //!<< most objects in use by client at one time
188     unsigned Allocations_;  //!<< total requests to allocate memory
189     unsigned Deallocations_; //!<< total requests to free memory
190 };
191
192 /**
193  * This allows us to easily treat raw objects as nodes in a linked list
194 */
195 struct GenericObject
196 {
197     GenericObject *Next; //!<< The next object in the list
198 };
199
200 /**
201  * This is used with external headers
202 */
203 struct MemBlockInfo
204 {
205     bool in_use;          //!<< Is the block free or in use?
206     char *label;          //!<< A dynamically allocated NUL-terminated string
207     unsigned alloc_num;   //!<< The allocation number (count) of this block
208 };
209
210 /**
211  * This class represents a custom memory manager
212 */
213 class ObjectAllocator
214 {

```

```
215 public:  
216     // Defined by the client (pointer to a block, size of block)  
217     typedef void (*DUMPCALLBACK)(const void *, size_t); //!< Callback function when dumping memory leaks  
218     typedef void (*VALIDATECALLBACK)(const void *, size_t); //!< Callback function when validating blocks  
219  
220     // Predefined values for memory signatures  
221     static const unsigned char UNALLOCATED_PATTERN = 0xAA; //!< New memory never given to the client  
222     static const unsigned char ALLOCATED_PATTERN = 0xBB; //!< Memory owned by the client  
223     static const unsigned char FREED_PATTERN = 0xCC; //!< Memory returned by the client  
224     static const unsigned char PAD_PATTERN = 0xDD; //!< Pad signature to detect buffer over/under flow  
225     static const unsigned char ALIGN_PATTERN = 0xEE; //!< For the alignment bytes  
226  
227     // Creates the ObjectManager per the specified values  
228     // Throws an exception if the construction fails. (Memory allocation problem)  
229     ObjectAllocator(size_t ObjectSize, const OAConfig& config);  
230  
231     // Destroys the ObjectManager (never throws)  
232     ~ObjectAllocator();  
233  
234     // Take an object from the free list and give it to the client (simulates new)  
235     // Throws an exception if the object can't be allocated. (Memory allocation problem)  
236     void *Allocate(const char *label = 0);  
237  
238     // Returns an object to the free list for the client (simulates delete)  
239     // Throws an exception if the object can't be freed. (Invalid object)  
240     void Free(void *Object);  
241  
242     // Calls the callback fn for each block still in use  
243     unsigned DumpMemoryInUse(DUMPCALLBACK fn) const;  
244  
245     // Calls the callback fn for each block that is potentially corrupted  
246     unsigned ValidatePages(VALIDATECALLBACK fn) const;  
247  
248     // Frees all empty pages  
249     unsigned FreeEmptyPages();  
250  
251     // Testing/Debugging/Statistic methods  
252     void SetDebugState(bool State); // true=enable, false=disable  
253     const void *GetFreeList() const; // returns a pointer to the internal free list  
254     const void *GetPageList() const; // returns a pointer to the internal page list  
255     OAConfig GetConfig() const; // returns the configuration parameters  
256     OAStats GetStats() const; // returns the statistics for the allocator  
257  
258     // Prevent copy construction and assignment  
259     ObjectAllocator(const ObjectAllocator &oa) = delete; //!< Do not implement!  
260     ObjectAllocator &operator=(const ObjectAllocator &oa) = delete; //!< Do not implement!  
261  
262 private:  
263  
264     OAConfig OAConfig_;  
265     OAStats OAStats_;  
266  
267     // Some "suggested" members (only a suggestion!)  
268     GenericObject *PageList_; //!< the beginning of the list of pages  
269     GenericObject *FreeList_; //!< the beginning of the list of objects  
270  
271     size_t midBlockSize;  
272     size_t offset;  
273  
274     //add page to the list  
275     void addPage();  
276     void updateOAStats();  
277  
278     size_t getPageSize();  
279     bool isEmpty(GenericObject* page) const;  
280     void freePage(GenericObject* temp);  
281     void removePgObjInFreeList(GenericObject* pageAddress);  
282     bool inPage(GenericObject* pageAddress, unsigned char* address) const;  
283  
284 };
```

## ObjectAllocator.cpp

```
1  ****  
2  /*!  
3  \file:      ObjectAllocator.cpp  
4  \author:    Goh Wei Zhe, weizhe.goh, 440000119  
5  \par email:  weizhe.goh@digipen.edu  
6  \date:     January 21, 2021  
7  \brief     To learn about and understand data structures and their  
8  ..... interfaces.  
9  
10 Copyright (C) 2021 DigiPen Institute of Technology.  
11 Reproduction or disclosure of this file or its contents without the  
12 prior written consent of DigiPen Institute of Technology is prohibited.  
13 */  
14 ****  
15  
16 #include "ObjectAllocator.h"  
17 #include <cstring>  
18  
19 #define u_char unsigned char  
20 ****  
21 /*!  
22 \fn      ObjectAllocator::ObjectAllocator(size_t ObjectSize,  
23 ..... const OAConfig& config)  
24  
25 \brief   Constructor to calculate and initialise class variables  
26  
27 \param  ObjectSize - the size of the object allocated  
28 \param  config - contains the OAConfig details for object allocator  
29 */  
30 ****  
31 ObjectAllocator::ObjectAllocator(size_t ObjectSize, const OAConfig& config)  
32 :OAConfig_{ config }, OAStats_{}, PageList_{ nullptr }, FreeList_{nullptr},  
33 midBlockSize{0}, offset{0}  
34 {  
35     OAStats_.ObjectSize_ = ObjectSize;  
36     OAStats_.PageSize_ = getPageSize();  
37  
38     //calculate size of block  
39     midBlockSize = OAConfig_.HBlockInfo_.size_ + OAConfig_.PadBytes_ * 2 +  
40     OAStats_.ObjectSize_ + OAConfig_.InterAlignSize_;  
41  
42     //calculate page size  
43     OAStats_.PageSize_ = sizeof(void*) + OAConfig_.LeftAlignSize_ +  
44     (OAConfig_.ObjectsPerPage_ * midBlockSize) - OAConfig_.InterAlignSize_;  
45  
46     //calculate offset to start at s  
47     //offset = P + al + h + s  
48     offset = sizeof(void*) + OAConfig_.LeftAlignSize_ +  
49 ..... OAConfig_.HBlockInfo_.size_ + OAConfig_.PadBytes_;  
50  
51     //if memory is allocated  
52     if(OAConfig_.UseCPPMemManager_){  
53         return;  
54     }  
55     //Page Allocation  
56     addPage();  
57 }  
58 ****  
59 /*!  
60 \fn      ObjectAllocator::~ObjectAllocator()  
61 \brief   Destructor for ObjectAllocator. Remove pages.  
62 */  
63 ****  
64 ObjectAllocator::~ObjectAllocator()  
65 {  
66     u_char* page;  
67  
68     while (PageList_ != NULL)  
69     {  
70         page = reinterpret_cast<u_char*>(PageList_);  
71  
72         PageList_ = PageList_->Next;  
73         delete [] page;  
74     }  
75 }  
76  
77 ****  
78 /*!  
79 \fn      ObjectAllocator::Allocate(const char* label)  
80  
81 \brief   Allocates memory for page and return to client. If UseCPPMemManager is  
82 ..... true, bypass OA and use new.  
83  
84 \param  label - the size of the object allocated  
85  
86 \return A pointer to a block of object  
87 */  
88 ****  
89 void* ObjectAllocator::Allocate(const char* label)  
90 {  
91     if(OAConfig_.UseCPPMemManager_){  
92         try  
93         {  
94             u_char* newObj = new u_char[OAStats_.ObjectSize_];  
95             updateOAStats();  
96  
97             return newObj;  
98         }  
99         catch (std::bad_alloc &)  
100        {  
101            throw OAException(OAException::E_NO_MEMORY, "Not enough memory!");  
102        }  
103    }  
104  
105    //Freelist is empty, no page allocated  
106  
107 }
```

```
108
109 if(FreeList_ == nullptr)
110 {
111     if(OAConfig_.MaxPages_ == 0 || OAStats_.PagesInUse_<OAConfig_.MaxPages_)
112     {
113         addPage();
114     }
115     else
116     {
117         throw OAEException(OAEException::E_NO_PAGES, "No available page!");
118     }
119 }
120
121 GenericObject* object = FreeList_;
122 FreeList_ = FreeList_->Next;
123
124 if(OAConfig_.DebugOn_)
125 {
126     memset(object, ALLOCATED_PATTERN, OAStats_.ObjectSize_);
127 }
128
129 updateOAStats();
130
131 switch(OAConfig_.HBlockInfo_.type_)
132 {
133     case OAConfig::hbBasic:
134     {
135         //Start of header
136         u_char* headerStart = reinterpret_cast<u_char*>(object) -
137             OAConfig_.PadBytes_ - OAConfig_.HBlockInfo_.size_;
138
139         unsigned* allocationNumber = reinterpret_cast<unsigned*>(headerStart);
140         *allocationNumber = OAStats_.Allocations_;
141
142         //Start of flag
143         u_char* flag = reinterpret_cast<u_char*>(headerStart +
144             sizeof(unsigned));
145
146         *flag = true;
147
148         break;
149     }
150     case OAConfig::hbExtended:
151     {
152         //Start of Flag
153         u_char* flagStart = reinterpret_cast<u_char*>(object) -
154             OAConfig_.PadBytes_ - sizeof(u_char);
155
156         *flagStart = true;
157
158         //Start of allocation number
159         u_char* allocNumStart = flagStart - sizeof(unsigned);
160
161         unsigned *allocationNumber =
162             reinterpret_cast<unsigned*>(allocNumStart);
163
164         *allocationNumber = OAStats_.Allocations_;
165
166         //Start of use-count
167         u_char* useCountStart = allocNumStart - sizeof(unsigned short);
168
169         unsigned short* useCount =
170             reinterpret_cast<unsigned short*>(useCountStart);
171
172         ++(*useCount);
173
174         break;
175     }
176     case OAConfig::hbExternal:
177     {
178         //one unsigned for allocation number, one flag to determine on off
179         u_char* headerStart = reinterpret_cast<u_char*>(object) -
180             OAConfig_.PadBytes_ - OAConfig_.HBlockInfo_.size_;
181
182         MemBlockInfo** memPtr =
183             reinterpret_cast <MemBlockInfo**>(headerStart);
184
185         try
186         {
187             *memPtr = new MemBlockInfo;
188
189             (*memPtr)->in_use = true;
190             (*memPtr)->label = nullptr;
191             (*memPtr)->alloc_num = OAStats_.Allocations_;
192
193             if(label)
194             {
195                 try
196                 {
197                     (*memPtr)->label = new char[strlen(label) + 1];
198
199                     catch (std::bad_alloc& )
200                     {
201                         throw OAEException(OAEException::E_NO_MEMORY,
202                             "Out of memory!");
203                     }
204
205                     strcpy((*memPtr)->label, label);
206                 }
207
208                 catch(std::bad_alloc& )
209                 {
210                     throw OAEException(OAEException::E_NO_MEMORY, "Out of memory!");
211                 }
212             }
213
214             default: break;
215         }
216     }
```

```
215     }
216 
217     return object;
218 }
219 
220 /**
221 *!
222 \fn void ObjectAllocator::Free(void* Object)
223 
224 \brief Free block memory from Allocate() If UseCPPMemManager is true, bypass
225 OA and use delete. If debug is on, perform check to see if block is
226 corrupted and within same boundaries.
227 
228 \param Object - A pointer to object allocated from Allocate().
229 */
230 
231 void ObjectAllocator::Free(void* Object)
232 {
233     if(OAConfig_.UseCPPMemManager_)
234     {
235         delete[] reinterpret_cast<u_char*>(Object);
236 
237         --OAStats_.ObjectsInUse_;
238         ++OAStats_.Deallocations_;
239         ++OAStats_.FreeObjects_;
240 
241         return;
242     }
243 
244     GenericObject* page = PageList_;
245     u_char* objAddress = reinterpret_cast<u_char*>(Object);
246 
247     //Only perform check when debugging
248     if(OAConfig_.DebugOn_)
249     {
250         bool onBound = false;
251         bool corrupted = false;
252         bool alreadyFreed = true;
253 
254         while(page != NULL)
255         {
256             if( reinterpret_cast<u_char*>(page) < objAddress &&
257                 objAddress < reinterpret_cast<u_char*>(page) + OAStats_.PageSize_ )
258             {
259                 u_char* blockStart = reinterpret_cast<u_char*>(page) + offset;
260 
261                 size_t separation = midBlockSize;
262 
263                 if((objAddress - blockStart) % separation == 0)
264                 {
265                     onBound = true;
266                 }
267             }
268 
269             page = page->Next;
270         }
271 
272         if(onBound == false)
273         {
274             throw(OAEException(OAEException::E_BAD_BOUNDARY,
275 "Object not on a block boundary"));
276         }
277         else
278         {
279             //check data block
280             for(unsigned i = sizeof(u_char*); i < OAStats_.ObjectSize_; ++i)
281             {
282                 if(objAddress[i] != FREED_PATTERN)
283                 {
284                     alreadyFreed = false;
285                     break;
286                 }
287             }
288 
289             //Check padding blocks
290             u_char* leftPad = objAddress - OAConfig_.PadBytes_;
291             u_char* rightPad = objAddress + OAStats_.ObjectSize_;
292 
293             for(unsigned int i = 0; i < OAConfig_.PadBytes_; ++i)
294             {
295                 if(leftPad[i] != PAD_PATTERN || rightPad[i] != PAD_PATTERN)
296                 {
297                     corrupted = true;
298                     break;
299                 }
300             }
301         }
302 
303         if(alreadyFreed == true)
304         {
305             throw(OAEException(OAEException::E_MULTIPLE_FREE,
306 "Block already freed once"));
307         }
308         else if (corrupted == true)
309         {
310             throw(OAEException(OAEException::E_CORRUPTED_BLOCK,
311 "Block is corrupted"));
312         }
313     }
314 
315     u_char* headerStart = reinterpret_cast<u_char*>(objAddress) -
316     OAConfig_.PadBytes_ - OAConfig_.HBlockInfo_.size_;
317 
318     switch(OAConfig_.HBlockInfo_.type_)
319     {
320         case OAConfig::hbBasic:
321         {
322             if(OAConfig_.HBlockInfo_.size_ > 0)
323             {
324                 OAConfig_.HBlockInfo_.size_--;
325             }
326         }
327     }
328 }
```

```
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
memset(headerStart, 0, OAConfig::BASIC_HEADER_SIZE);
break;
}
case OAConfig::hbExtended:
{
    memset(headerStart + OAConfig_.HBlockInfo_.additional_ +
sizeof(unsigned short), 0, OAConfig::BASIC_HEADER_SIZE);

    break;
}
case OAConfig::hbExternal:
{
    MemBlockInfo** memPtr =
reinterpret_cast<MemBlockInfo*>(headerStart);

    delete[] (*memPtr)->label;
    delete *memPtr;

    memset(memPtr, 0, OAConfig_.HBlockInfo_.size_);

}
default: break;
}

if(OAConfig_.DebugOn_)
{
    memset(Object, FREED_PATTERN, OAStats_.ObjectSize_);
}

GenericObject* object = reinterpret_cast<GenericObject*>(Object);
object->Next = FreeList_;
FreeList_ = object;

--OAStats_.ObjectsInUse_;
++OAStats_.Deallocations_;
++OAStats_.FreeObjects_;

}

/*****************************************/
/*!
\fn     unsigned ObjectAllocator::DumpMemoryInUse(DUMPCALLBACK fn) const
\brief Run through pages in OA and calls the given function if the block
       data are active.
\param fn - A function to call for each active object.
\return Number of active objects
*/
/*****************************************/
unsigned ObjectAllocator::DumpMemoryInUse(DUMPCALLBACK fn) const
{
    GenericObject* page = PageList_;

    u_char* block;

    while (page)
    {
        block = reinterpret_cast<u_char*>(page) + offset;

        for (unsigned i = 0; i < OAConfig_.ObjectsPerPage_; ++i)
        {
            if (*block == ALLOCATED_PATTERN)
            {
                fn(block, OAStats_.ObjectSize_);
            }
            if (i < OAConfig_.ObjectsPerPage_ - 1)
            {
                block += midBlockSize;
            }
        }

        page = page->Next;
    }

    return OAStats_.ObjectsInUse_;
}

/*****************************************/
/*!
\fn     unsigned ObjectAllocator::ValidatePages(VALIDATECALLBACK fn) const
\brief Run through each pages to check if each data block is corrupted or not.
\param fn - A function to call once a data block found corrupted.
\return Number of corrupted data blocks.
*/
/*****************************************/
unsigned ObjectAllocator::ValidatePages(VALIDATECALLBACK fn) const
{
    if (OAConfig_.PadBytes_ == 0)
    {
        return 0;
    }

    unsigned corruptedBlocks = 0;
    GenericObject* page = PageList_;

    u_char* block;
    u_char* leftPad;
    u_char* rightPad;

    while (page)
    {
        block = reinterpret_cast<u_char*>(page) + offset;

        for (unsigned i = 0; i < OAConfig_.ObjectsPerPage_; ++i)
```

```
429  {
430     // Check padding blocks
431     leftPad = block - OAConfig_.PadBytes_;
432     rightPad = block + OAStats_.ObjectSize_;
433
434     for (unsigned j = 0; j < OAConfig_.PadBytes_; ++j)
435     {
436         if (leftPad[j] != PAD_PATTERN || rightPad[j] != PAD_PATTERN)
437         {
438             fn(block, OAStats_.ObjectSize_);
439             ++corruptedBlocks;
440             break;
441         }
442     }
443
444     // Prevents pointer arithmetic overrun
445     if (i < OAConfig_.ObjectsPerPage_ - 1)
446     {
447         block += midBlockSize;
448     }
449
450     page = page->Next;
451 }
452
453     return corruptedBlocks;
454 }
455
456 /*!
457 \fn     unsigned ObjectAllocator::FreeEmptyPages()
458
459 \brief Run through each page to check if page is empty. If empty, returns
460       memory back to OS.
461
462 \return Number of pages freed.
463 */
464
465 /*!
466 \fn     unsigned ObjectAllocator::FreeEmptyPages()
467
468     if(PageList_ == nullptr)
469         return 0;
470
471     unsigned emptyPages = 0;
472
473     GenericObject* temp = PageList_;
474     GenericObject* previous = nullptr;
475
476     //If head node holds a key or multiple key
477     while(temp != nullptr && isPageEmpty(temp))
478     {
479         //Change head
480         PageList_ = temp->Next;
481         freePage(temp);
482
483         //Change temp
484         temp = PageList_;
485         ++emptyPages;
486     }
487
488     //Delete occurrence other than head
489     while(temp != nullptr)
490     {
491         //Search key to be deleted
492         //Keep track of previous node as we need to change previous->Next
493         while(temp != nullptr && !isPageEmpty(temp))
494         {
495             previous = temp;
496             temp = temp->Next;
497         }
498
499         //If key not present in linked list
500         if(temp == nullptr)
501             return emptyPages;
502
503         //Unlink node from linked list
504         previous->Next = temp->Next;
505         freePage(temp);
506
507         //Update temp for next loop
508         temp = previous->Next;
509         ++emptyPages;
510     }
511
512     return emptyPages;
513 }
514
515 /*!
516 \fn     void ObjectAllocator::SetDebugState(bool State)
517
518 \brief Set current OA to the given debug state.
519
520 \param State - Debug on (True), Debug off (False)
521 */
522
523 /*!
524 \fn     void ObjectAllocator::SetDebugState(bool State)
525
526     OAConfig_.DebugOn_ = State;
527 }
528
529 */
530 /*!
531 \fn     const void* ObjectAllocator::GetFreeList() const
532
533 \brief Returns the free list, a linked list to determine which block of memory
534       are free.
535 
```

```
536     \return The free list.  
537  
538     */  
539     /*********************************************************************/  
540     const void* ObjectAllocator::GetFreeList() const  
541     {  
542         return FreeList_;  
543     }  
544  
545     /*********************************************************************/  
546     /*!  
547     \fn     const void* ObjectAllocator::GetPageList() const  
548  
549     \brief Returns the page list, a linked list that chains all the page.  
550     .....  
551     \return The page list.  
552  
553     */  
554     /*********************************************************************/  
555     const void* ObjectAllocator::GetPageList() const  
556     {  
557         return PageList_;  
558     }  
559  
560     /*********************************************************************/  
561     /*!  
562     \fn     OAConfig ObjectAllocator::GetConfig() const  
563  
564     \brief Returns a copy of the current configuration.  
565  
566     \return The copy of the current configuration  
567  
568     */  
569     /*********************************************************************/  
570     OAConfig ObjectAllocator::GetConfig() const  
571     {  
572         return OAConfig_;  
573     }  
574  
575     /*********************************************************************/  
576     /*!  
577     \fn     OAStats ObjectAllocator::GetStats() const  
578  
579     \brief Returns a copy of the current stats.  
580  
581     \return The copy of the current stats  
582  
583     */  
584     /*********************************************************************/  
585     OAStats ObjectAllocator::GetStats() const  
586     {  
587         return OAStats_;  
588     }  
589  
590     /*********************************************************************/  
591     /*!  
592     \fn     void ObjectAllocator::addPage()  
593  
594     \brief Allocate new page. Checks if it has hit the maximum page allocated  
595     ..... allowed to allocate.  
596     */  
597     /*********************************************************************/  
598     void ObjectAllocator::addPage()  
599     {  
600         //Check if maximum pages is reached  
601         if(OAConfig_.MaxPages_ && (OAStats_.PagesInUse_ == OAConfig_.MaxPages_))  
602         {  
603             throw OAEException(OAEException::E_NO_PAGES, "maximum page allocated");  
604         }  
605  
606         GenericObject* newPassword;  
607  
608         try  
609         {  
610             //allocate memory for new page  
611             newPassword =  
612                 reinterpret_cast<GenericObject*>(new u_char[OAStats_.PageSize_]);  
613  
614             ++OAStats_.PagesInUse_;  
615         }  
616         catch(std::bad_alloc &)  
617         {  
618             throw OAEException(OAEException::E_NO_PAGES,  
619             "Not enough memory to add new page");  
620         }  
621  
622         //link PageList_  
623         newPassword->Next = PageList_;  
624         PageList_ = newPassword;  
625  
626         //Putting object onto free list  
627         unsigned char* temp = reinterpret_cast<unsigned char*>(newPage);  
628         temp += sizeof(void*);  
629  
630         memset(temp, ALIGN_PATTERN, OAConfig_.LeftAlignSize_);  
631  
632         temp += (offset - sizeof(void*));  
633  
634         for(unsigned int i = 0; i < OAConfig_.ObjectsPerPage_; ++i)  
635         {  
636             ++OAStats_.FreeObjects_;  
637  
638             if(OAConfig_.DebugOn_)  
639             {  
640                 .....  
641                 memset(temp - OAConfig_.PadBytes_ - OAConfig_.HBlockInfo_.size_, 0,  
642                         OAConfig_.HBlockInfo_.size_);  
643             }  
644         }  
645     }
```

```
643     memset(temp, UNALLOCATED_PATTERN, OAStats_.ObjectSize_);
644     memset(temp - OAConfig_.PadBytes_, PAD_PATTERN, OAConfig_.PadBytes_);
645     memset(temp + OAStats_.ObjectSize_, PAD_PATTERN, OAConfig_.PadBytes_);
646 }
647
648 if(i != OAConfig_.ObjectsPerPage_ - 1)
649 {
650     memset(reinterpret_cast<unsigned char*>(temp) + OAStats_.ObjectSize_
651         + OAConfig_.PadBytes_, ALIGN_PATTERN, OAConfig_.InterAlignSize_);
652 }
653
654 reinterpret_cast<GenericObject*>(temp)->Next = FreeList_;
655 FreeList_ = reinterpret_cast<GenericObject*>(temp);
656
657 temp += OAStats_.ObjectSize_ + 2* OAConfig_.PadBytes_ +
658 OAConfig_.HBlockInfo_.size_ + OAConfig_.InterAlignSize_;
659 }
660 }
661
662 ****
663 */
664 \fn void ObjectAllocator::updateOAStats()
665
666 \brief Increment stats every time an object is allocated.
667 */
668 ****
669 void ObjectAllocator::updateOAStats()
670 {
671     ++OAStats_.Allocations_;
672     ++OAStats_.ObjectsInUse_;
673
674     if(OAStats_.ObjectsInUse_ > OAStats_.MostObjects_)
675         OAStats_.MostObjects_ = OAStats_.ObjectsInUse_;
676
677     --OAStats_.FreeObjects_;
678 }
679
680 ****
681 */
682 \fn size_t ObjectAllocator::getPageSize()
683
684 \brief Returns the page size
685
686 \return The page size
687 */
688 ****
689 size_t ObjectAllocator::getPageSize()
690 {
691     //check if left alignment needs to be calculated
692     if(OAConfig_.Alignment_ > 1)
693     {
694         //calculate byte needed for left alignment
695         size_t left_bytes = sizeof(void*) + OAConfig_.HBlockInfo_.size_ +
696             OAConfig_.PadBytes_;
697
698         OAConfig_.LeftAlignSize_ = OAConfig_.Alignment_ -
699             (static_cast<unsigned>(left_bytes) % OAConfig_.Alignment_);
700
701         //calculate byte needed for inter alignment
702         size_t inter_bytes = OAStats_.ObjectSize_ + OAConfig_.HBlockInfo_.size_
703             + OAConfig_.PadBytes_ * 2;
704
705         OAConfig_.InterAlignSize_ = OAConfig_.Alignment_ -
706             (static_cast<unsigned>(inter_bytes) % OAConfig_.Alignment_);
707     }
708
709     size_t size = 0;
710
711     //left alignment size
712     //nextP + al (left alignment). Only first block has left alignment
713     size = sizeof(void*) + OAConfig_.LeftAlignSize_;
714
715     //InterAlignSize_
716     size += OAConfig_.ObjectsPerPage_* (OAConfig_.HBlockInfo_.size_ +
717         OAConfig_.PadBytes_ + OAStats_.ObjectSize_ + OAConfig_.PadBytes_);
718
719     size += (OAConfig_.ObjectsPerPage_ - 1) * OAConfig_.InterAlignSize_;
720
721     return size;
722 }
723
724 ****
725 */
726 \fn bool ObjectAllocator::isPageEmpty(GenericObject* page) const
727
728 \brief Given a page, check if the page is empty by running through the
729         freelist and check if there are free items in the page.
730
731 \param page - A page to check if it is empty
732
733 \return Return true if page is empty, otherwise, return false.
734 */
735 ****
736 bool ObjectAllocator::isPageEmpty(GenericObject* page) const
737 {
738     GenericObject* freeList = FreeList_;
739
740     unsigned freeWithinPage = 0;
741
742     while(freeList)
743     {
744         if(inPage(page, reinterpret_cast<u_char*>(freeList)))
745         {
746             ++freeWithinPage;
747
748             if(freeWithinPage >= OAConfig_.ObjectsPerPage_)
749             {
750                 return true;
751             }
752         }
753     }
754
755     return false;
756 }
```

```
750     ....      return true;
751   }
752 }
753 freeList = freeList->Next;
754 }
755 return false;
756 }
757 ****
758 */
759 */
760 \fn void ObjectAllocator::freePage(GenericObject* temp)
761 \brief Given a page, free memory by returning it to OS. Removes all data block
762     from the freelist, then decrement the stat counter.
763
764 \param temp - A page to free.
765 */
766 ****
767 */
768 void ObjectAllocator::freePage(GenericObject* temp)
769 {
770     removePgObjInFreeList(temp);
771     delete[] reinterpret_cast<u_char*>(temp);
772     --OAStats_.PagesInUse_;
773 }
774
775 void ObjectAllocator::removePgObjInFreeList(GenericObject* pageAddress)
776 {
777     GenericObject* temp = FreeList_;
778     GenericObject* previous = nullptr;
779
780     //If head node itself hold key or multiple key
781     while(temp!= nullptr && inPage(pageAddress,reinterpret_cast<u_char*>(temp)))
782     {
783         //Change head
784         FreeList_ = temp->Next;
785
786         //Change temp
787         temp = FreeList_;
788         --OAStats_.FreeObjects_;
789     }
790
791     //Delete occurence other than head
792     while(temp != NULL)
793     {
794         //Search key to be deleted
795         //Keep track of previous node, need to change previous->Next
796         while(temp != nullptr &&
797             !inPage(pageAddress, reinterpret_cast<u_char*>(temp)))
798         {
799             previous = temp;
800             temp = temp->Next;
801         }
802
803         //If key not present in linked list
804         if(temp == nullptr) return;
805
806         //unlink node from linked list
807         previous->Next = temp->Next;
808
809         --OAStats_.FreeObjects_;
810
811         //Update temp for next loop
812         temp = previous->Next;
813     }
814 }
815 ****
816 */
817 */
818 \fn bool ObjectAllocator::inPage(GenericObject* pageAddress,
819     u_char* address) const
820
821 \brief Given a page address and any address, check if address is in the page.
822
823 \param pageAddress - The pointer to the page.
824 \param address - The address to check if it is in range.
825
826 \return Returns true if in page, otherwise, return false.
827 */
```

index.chm





196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

276

277

278

279

280

281

282

283

284

285

286

287

288

289

290

291

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310

311

312

313

314

315

316

317

318

319

320

321















1071 .\ \ 1072 à\ \ 1073 #\ \ 1074 à\ \ 1075 ä! \ 1076 a\ \ 1077 Iu\ \ 1078 öñ\ \ 1079 .@ \ 1080 <\ \ 1081 Å\ \ 1082 E\ \ 1083 'a!\ \ 1084 IÖ\ \ 1085 %E\ \ 1086 UVX\ \ 1087 Y^"\ \ 1088 n\ \ 1089 \$WU\ \ 1090 Å\ \ 1091 ,\ \ 1092 :~M#1\ \ 1093 ¥V\ \ 1094 J'\ \ 1095 ' /ä\ \ 1096 m\ \ 1097 ÅUJ\ \ 1098 È\ \ 1099 S\ \ 1100 -:\ \ 1101 éil\ \ 1102 o\ \ 1103 ÄY\ \ 1104 0c\ \ 1105 G\ \ 1106 D\ \ 1107 «\ \ 1108 \*Y\ \ 1109 <aiC\ \ 1110 'ÄG\ \ 1111 éo\*\ \ 1112 o\ \ 1113 ¥D\ \ 1114 c\ \ 1115 å\ \ 1116 =P\ \ 1117 V3\ \ 1118 \ \ 1119 \ \ 1120 %\ \ 1121 é\ \ 1122 b\ \ 1123 y\ \ 1124 ug\ \ 1125 i\ \ 1126 a\ \ 1127 M\ \ 1128 I\ \ 1129 dZ\ \ 1130 Ü\ \ 1131 z\ \ 1132 Oj\ \ 1133 .46\ \ 1134 á\ \ 1135 \ \ 1136 X\ \ 1137 ,\ \ 1138 8\ \ 1139 Å\ \ 1140 \ \ 1141 Ö\ \ 1142 \ \ 1143 \ \ 1144 j4\ \ 1145 Q6\ \ 1146 "i\ \ 1147 9\ \ 1148 é\ \ 1149 %\ \ 1150 i\ \ 1151 \ \ 1152 S7\ \ 1153 A\ \ 1154 u\ \ 1155 ÜF\ \ 1156 1\ \ 1157 \ \ 1158 \ \ 1159 Z\ \ 1160 HA\ \ 1161 y9\ \ 1162 p\ \ 1163 å7\ \ 1164 h\_g<\ \ 1165 ä\ \ 1166 \ \ 1167 'I\ \ 1168 \ \ 1169 ?S\ \ 1170 ÜD\ \ 1171 à\ \ 1172 k\ \ 1173 R\ \ 1174 \ \ 1175 \ \ 1176 \ \ 1177







VPL

## **◀ General Information**

## Jump to...

Introduction to Data Structures ►

You are logged in as Wei Zhe GOH ([Log out](#))

cs280s21-b.sg

## Data retention summary

[Get the mobile app](#)